KubeCon CloudNativeCon North America 2026

#KubeCon #CloudNativeCon

Splitting the Monolith at Cloudera

Building Fault-Tolerant Workflows with Cadence



Ender Demirkaya
Uber, Cadence

Soma Koltai
Cloudera

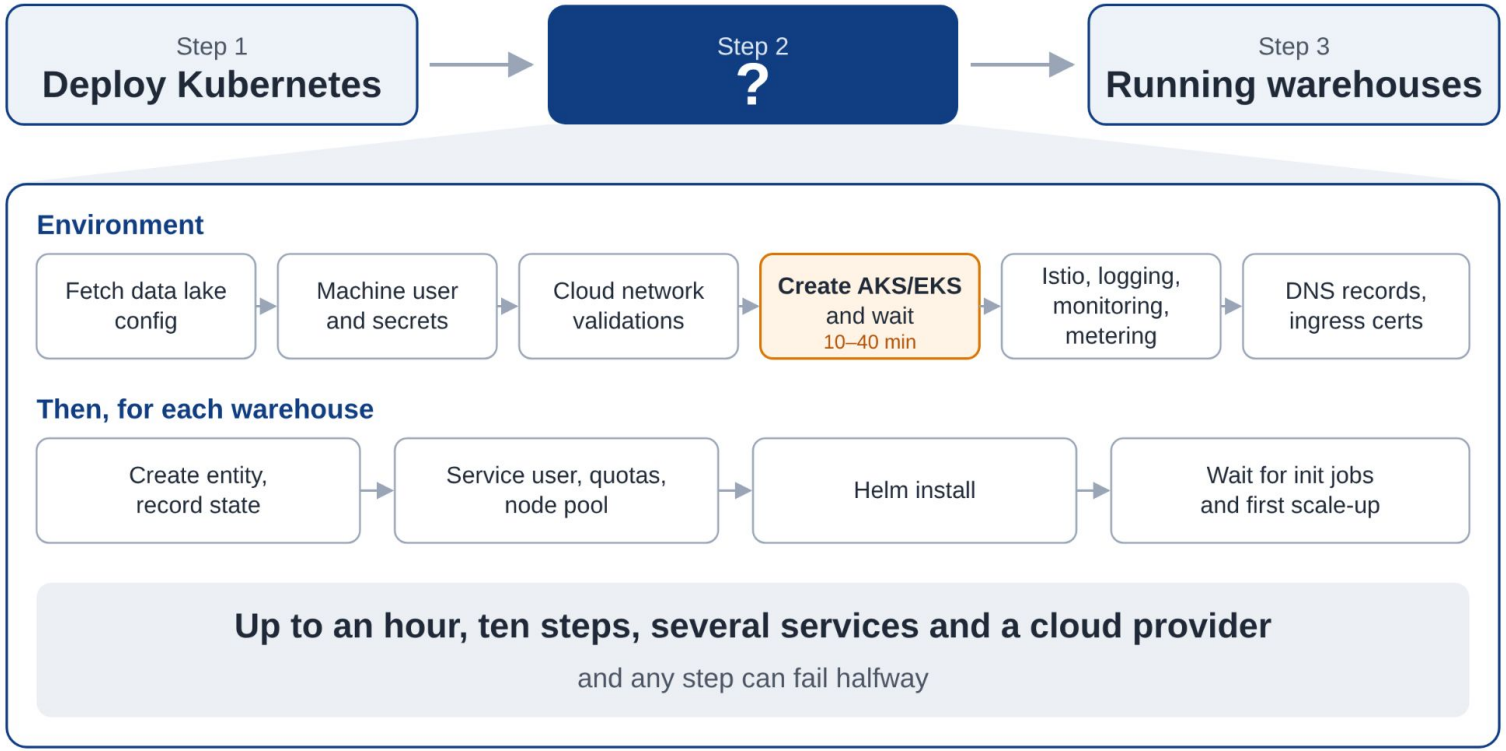
Agenda

- CDW (Cloudera Data Warehouse) overview
- Why we chose Cadence
- Cadence overview
- Bridging the gap: Async Executor
- Lease API: locking across old and new code
- Transitioning to Cadence
- What we learned (examples/stats)

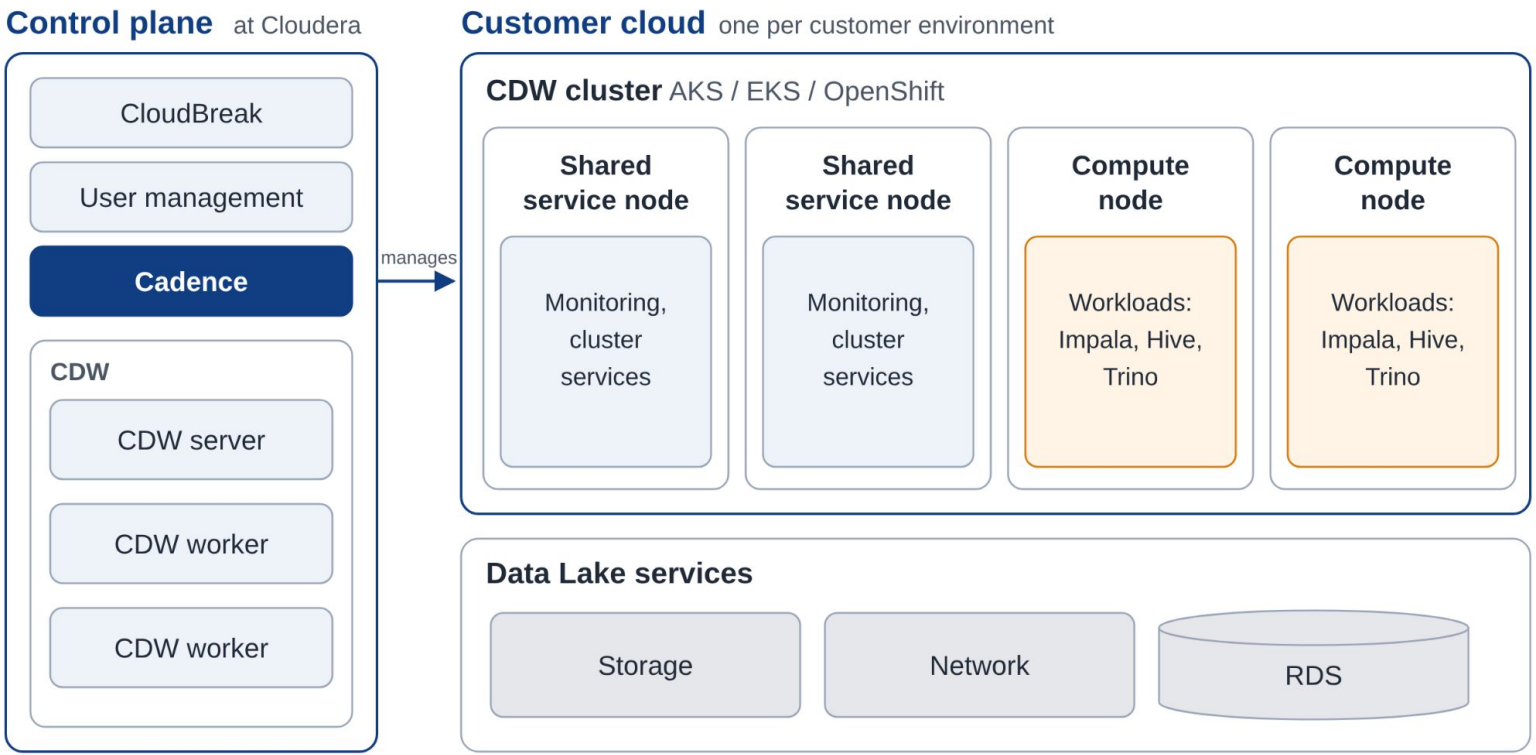
CDW Overview

- Service to deploy Database Catalog, Impala, Hive, Trino to Kubernetes clusters
- Managed K8s on public cloud (AKS, EKS), customer managed OpenShift or RKE2 for private cloud
- Environment creation
  - Deploy shared components and prerequisites (CNI, storage drivers, monitoring)
  - Deploy Database Catalog
- Create workloads
  - Manage node pool assignments
  - Lifecycle management (CRUD + health and status checks - similar to k8s operators)

CDW Overview

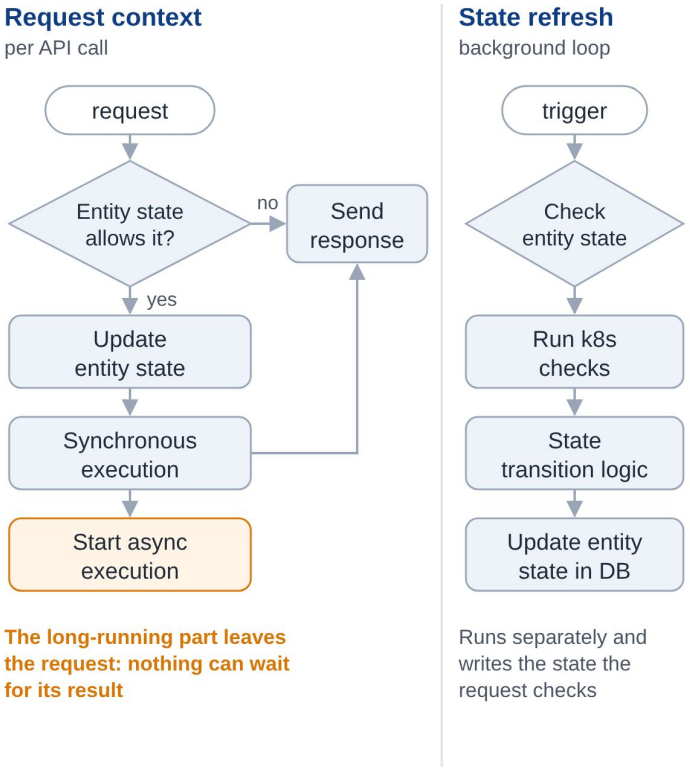


Control plane - Customer Env / Data Lake



Previous architecture

- Single server
- Monolithic architecture
- State based entity locking
- State transitions done in a separate context
- Long running tasks
  - Cluster creation 10-40 mins
  - Workload CRUD ops 1-20 mins



The long-running part leaves the request: nothing can wait for its result

Runs separately and writes the state the request checks

Why we needed a workflow engine?

- Performance issues raised the question of scalability
- No fault tolerance (server restart)
- Long running tasks (20-40 mins)
  - Hard to implement more complex features (environment rebuild, backup/restore)
- Requirements
  - Distributed
  - Resilient, fault tolerant
  - High Availability
  - Improved observability

Why Cadence?

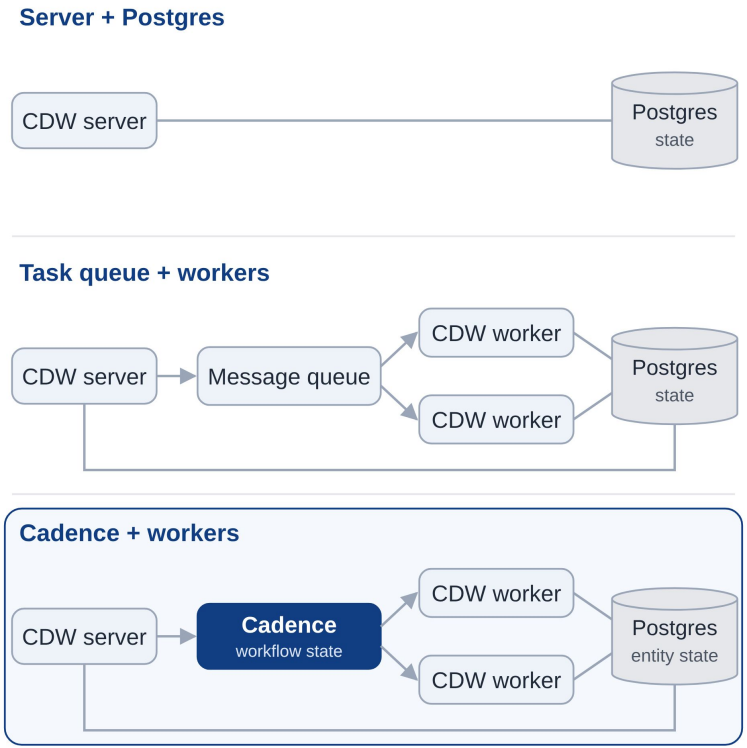
Custom task queue

- Solves horizontal scalability
- Improves reliability

But we'd have to implement*

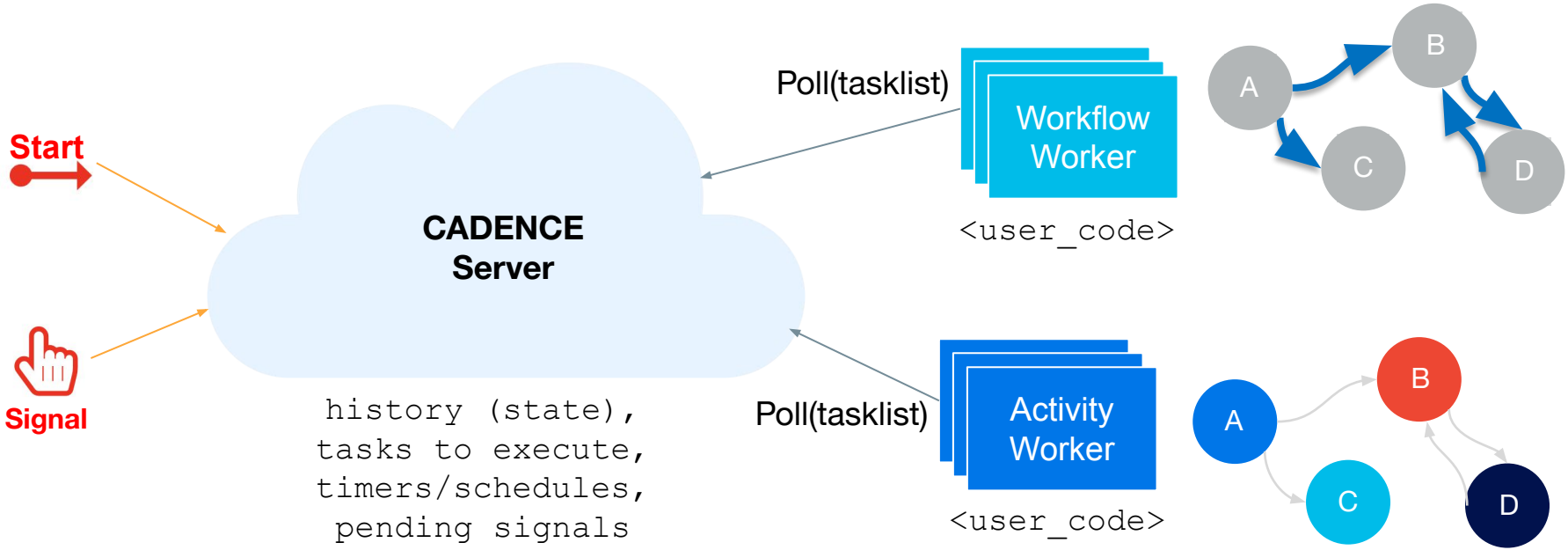
- Automatic retries, rescheduling
- Worker health check
- Advanced features: awaitable results, context serialization and propagation, etc
- A developer friendly API/Client

*and maintain it



Cadence Architecture

Orchestration is managed by a Cadence server while your service only contains your code



Key Cadence Concepts

KubeCon CloudNativeCon North America 2026

Domain →	**Account** for team/project Has its own **Quota / Isolation**
Workflow →	**Use case**
Activity	**Functional Execution Unit** (Single Workflow Step)
Tasklist →	Execution Pools **Prioritization / Rate Limits**

Key Cadence Features

- Activity & Workflow
  - **Failures** are **retried**
  - **Inputs & outputs** are **recorded**
- Workflows **resume** while workers **restart** or **crash**
- Workflow executions are
  - **Deterministic** => **replayable**
  - **Durable**

What Cadence brings, and what we added

Cadence brings

- Distributed execution and horizontal scaling
- Fault tolerance: HA for business-critical workflows
- Kubernetes-native deployment
- Workflows for logic, activities for side effects
- Testability and observability

What we had to add

- **Leasing: distributed locking**
- Context propagators
- Logging integration
- Admin tools: lease management, audit events
- Our own workflows and activities

Bridging the Gap: Async Executor

- Adopting Cadence will take some time, but we needed a solution that can be used as a stand-in during development
- Key requirements
  - executions are started in the background, the result can be fetched by a blocking call.
  - Context, arguments and return data is **serialized** in the process
- Modeled after Cadence client library
- Implements a subset of Cadence functionality and API
  - Activity execution
    - Provisional response (Future)
    - Context propagation
  - Similar API signature

Async executor

What to implement?


```
func init() {
	activity.Register(SimpleActivity)
}

func SimpleWorkflow(ctx workflow.Context, value string) error {
	ao := workflow.ActivityOptions{
		TaskList:               "sampleTaskList",
		ScheduleToCloseTimeout: time.Second * 60,
		ScheduleToStartTimeout: time.Second * 60,
		StartToCloseTimeout:    time.Second * 60,
		HeartbeatTimeout:       time.Second * 10,
		WaitForCancellation:    false,
	}
	ctx = workflow.WithActivityOptions(ctx, ao)

	future := workflow.ExecuteActivity(ctx, SimpleActivity, value)
	var result string
	if err := future.Get(ctx, &result); err != nil {
		return err
	}
	workflow.GetLogger(ctx).Info("Done", zap.String("result", result))
	return nil
}
```

AsyncExec / Same code shape as Cadence

Cadence

```
func SimpleWorkflow(ctx workflow.Context, v string) (string, error) {
    ctx = workflow.WithActivityOptions(ctx, opts)
    future := workflow.ExecuteActivity(ctx, SimpleActivity, v)
    var res string
    err := future.Get(ctx, &res)
    return res, err
}
```

AsyncExec

```
func SimpleWorkflow(ctx context.Context, v string) (string, error) {
    ctx = WithExecutionTimeout(ctx, time.Second)
    future := workflow.Execute(ctx, "SimpleActivity", v)
    var res string
    err := future.Get(ctx, &res)
    return res, err
}
```

Transitioning to Cadence

We had some things to do first

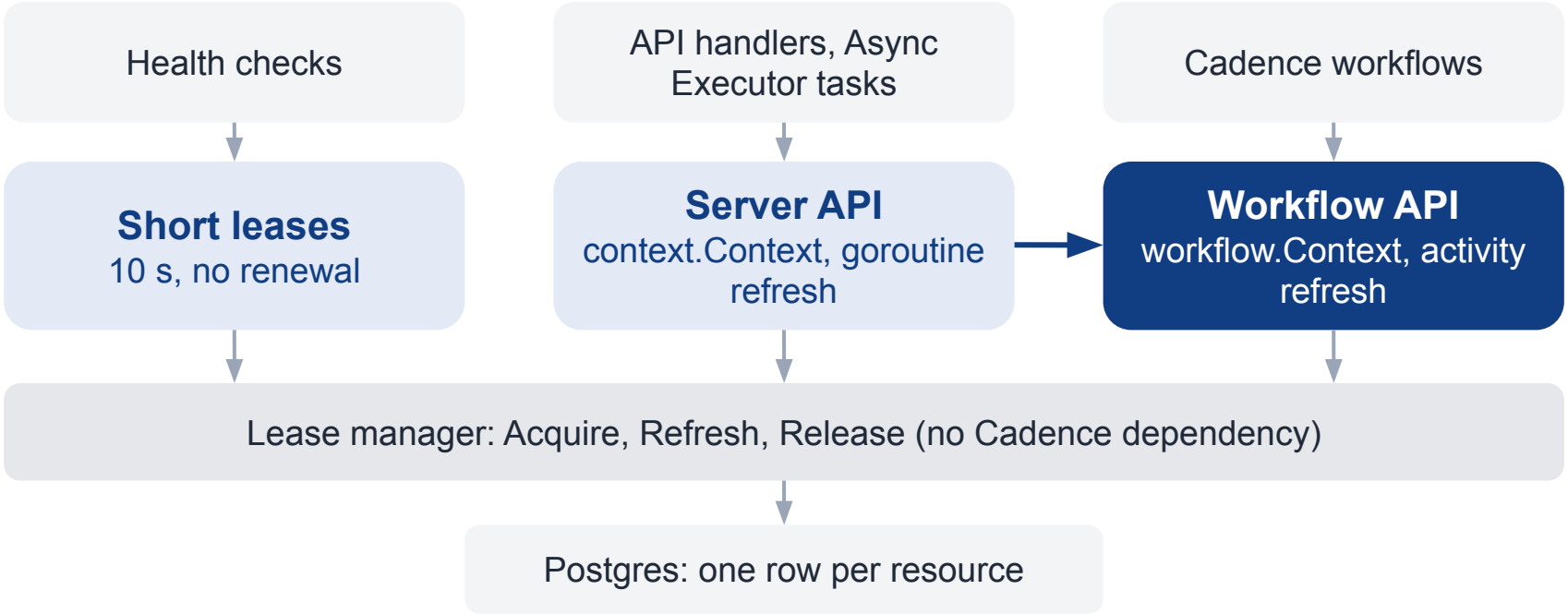
- Distributed locking
- In-memory caches
- Stateful global components
- Dependency injection
- Lease API

From locks to leases

- Before: entity status and DB transactions that spanned slow cloud calls
- Many servers and workers: the lock must live outside any one process
- Try-lock: of two conflicting operations, one fails within seconds
- A worker can die mid-operation, so assume the unlock never happens
- **Lease = lock + TTL**: 5 min TTL, renewed every minute, 5 h hard cap

```
type LeaseAPI interface {
      Acquire(resource, operationID string, ttl, wait time.Duration) (lease ID, err error)
      Refresh(leaseID ID, ttl time.Duration) (err error)
      Release(leaseID ID) error
}
```

Lease API layers



Lease API: acquiring a lease

KubeCon CloudNativeCon North America 2026

```
INSERT INTO lease (entity_id, operation_id, expiry_time, instance_id)
VALUES ($1, $2, $3, $4)
ON CONFLICT (entity_id) DO NOTHING
RETURNING id;   -- no row back: someone else holds it
```

- A unique key on the resource does the mutual exclusion: no row locks, no advisory locks
- Holder expired: delete it by lease ID and retry, so two waiters can't remove each other's fresh lease
- Same operation already holds it: error, not re-entry
- Otherwise poll until the wait period ends (default 10 s), then fail fast
- Nothing reaps expired rows; the next acquire clears them

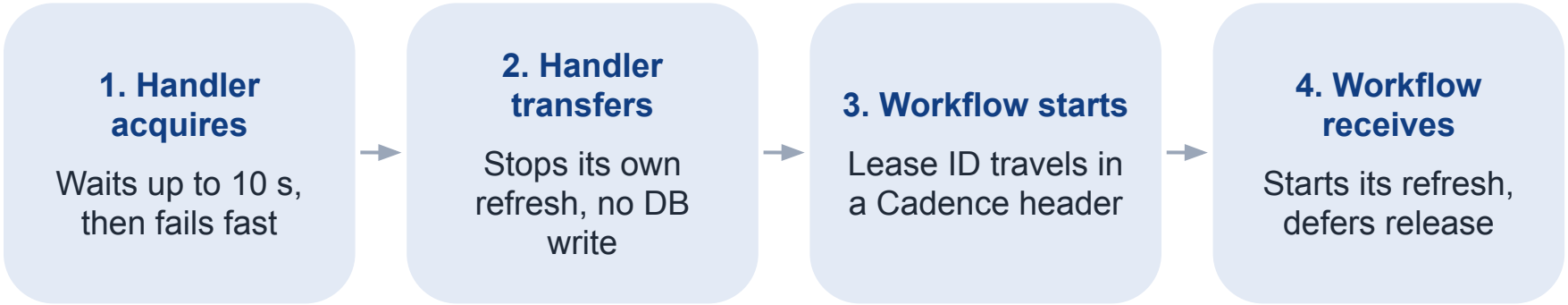
Lease API in Cadence workflows


```
// The lease lives in workflow.Context
type WorkflowAPI interface {
    Acquire(ctx workflow.Context, entityID string,
        opts ...AcquireWfOptionFunc) (workflow.Context, error)
    Receive(ctx workflow.Context, entityID string) (workflow.Context, error)
    Release(ctx workflow.Context, entityID string) error
}

func UpdateWorkflow(ctx workflow.Context, in UpdateInput) error {
    ctx, err := leaseAPI.Receive(ctx, in.WarehouseID) // handed over by the API call
    if err != nil {
        return err
    }
    defer leaseAPI.Release(ctx, in.WarehouseID) // runs on completion, failure, cancellation
    return workflow.ExecuteActivity(ctx, UpdateActivity, in).Get(ctx, nil)
}
```

- A heartbeated refresh activity renews the lease; if it stops, the context is cancelled
- Release runs on a disconnected context, so it survives cancellation

Lease API: handing over a lease



- Old and new code share one lease: the API call fails fast, and the workflow inherits the lease instead of racing for it
- About 20 workflows start this way

Lease API: when things fail

Event	What happens to the lease
Worker pod rescheduled mid-workflow	Workflow replays elsewhere; the refresh activity is retried on another worker
Server pod dies holding a lease	Its refresh goroutine dies too; the lease expires within 5 min
Workflow cancelled or failed	Deferred release still runs
Worker dies during acquire	Workflow fails; the row is orphaned until its TTL (5 min at most)
A delete arrives mid-operation	Cancel the workflow holding the lease and take it over; force-acquire as fallback

Distributed Semaphores - Workflows
(Coming Soon)

KubeCon CloudNativeCon North America 2026

```
// Using the semaphore (caller starts the workflow):
client.StartWorkflow(ctx, client.StartWorkflowOptions{
   ID:          "training-job-456",
   TaskList:    "gpu-workers",
   SemaphoreID:    "gpu-pool",  // only 10 workflows can run at once
}, MySampleWorkflow, modelID)
```

```
// Defining the semaphore
// Admin setup (one-time, server-side)
cadenceAdmin.CreateSemaphore(ctx, &admin.CreateSemaphoreRequest{
   Domain:      "ml-platform",
   SemaphoreID: "gpu-pool",
   Capacity:    10,
})
```

Distributed Semaphores - Activities
(Coming Soon)


```
// Activity 2: semaphore limits concurrent Spark calls to 100
ctx2 := workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
    TaskList:                "spark-workers",
    StartToCloseTimeout:     time.Minute,
    ScheduleToStartTimeout: 5 * time.Minute, // also bounds the permit wait
    SemaphoreID:             "spark-jobs-limited",
})
err := workflow.ExecuteActivity(ctx2, aggregatePayments).Get(ctx, nil)
```

Distributed Semaphores - Code Block
(Coming Soon)


```
sem := workflow.GetSemaphore(ctx, "mySem")

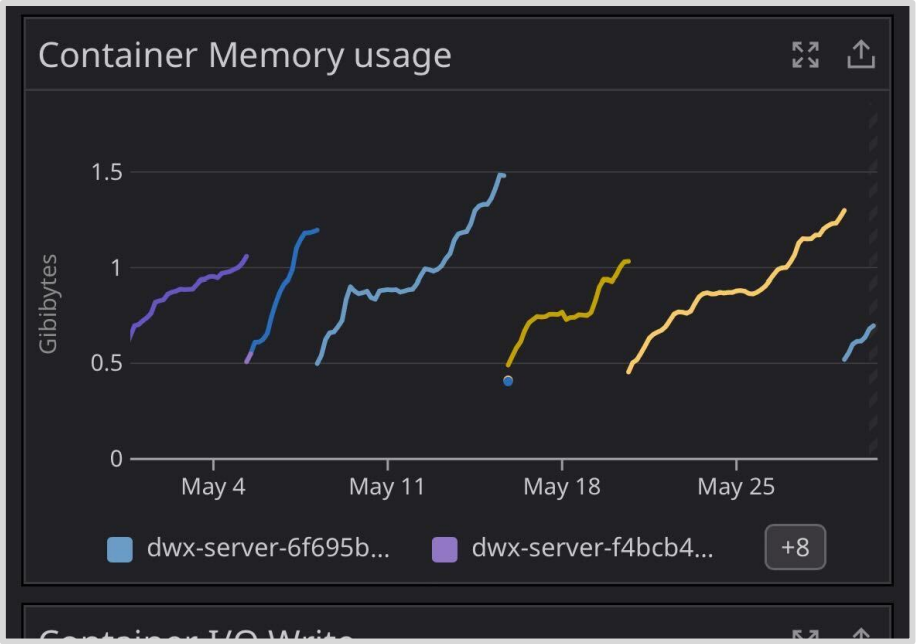
// Phase 2: blocked
// ONE acquire() = ONE hold = ONE token, and it covers the WHOLE block
if err := sem.Acquire(workflow.WithAcquireTimeout(ctx, time.Minute)); err != nil {
    // acquire timeout, semaphore doesn't exist, ...
    // handle error
}
f1 := workflow.ExecuteActivity(ctx, activity1, 1)
f2 := workflow.ExecuteActivity(ctx, activity2, 2)
err1 := f1.Get(ctx, nil)
err2 := f2.Get(ctx, nil)

// return the token as soon as the guarded work is done
sem.Release(ctx)
```

Proof of stability

The memory leak that exposed the restarts

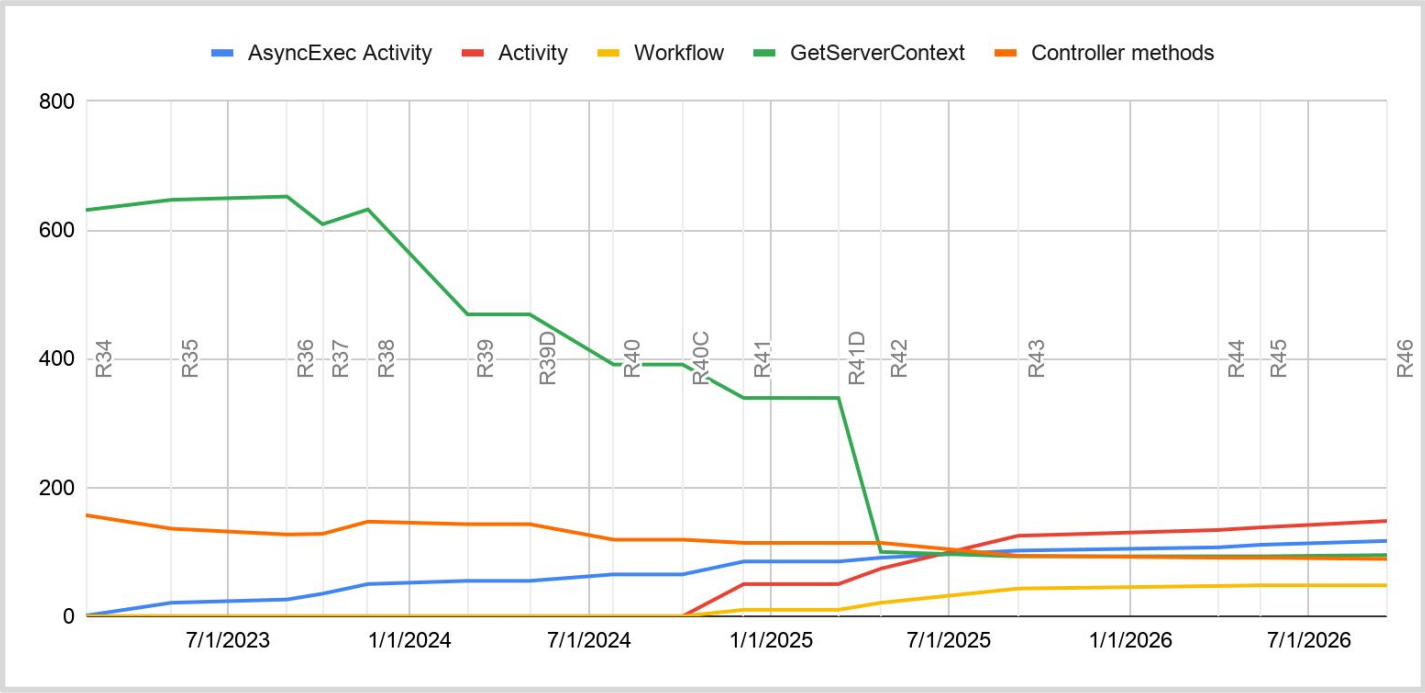
- During the investigation of a memory leak we found unusual restarts unrelated to the memory leak
- Server restarts previously caused issues like cluster provisioning failures or workload upgrade failures
- The SRE uses Karpenter on the clusters to defragment the nodes, which causes the rescheduling of server pods **before** the OOM limit is hit



What we learned

- The transition from AsyncExec and Lease API to Cadence and Workflow API went as expected 🎉
- Cadence can reveal the cracks in architecture and implementation
- Not just another external service - it affects architectural and implementation decisions
- Acquiring a lease is the one step you can't make idempotent
- Keep-alive belongs in a heartbeated activity; release must survive cancellation
- Improved testability
- Improved code quality
  - Even without Cadence, just by applying the patterns

Code stats


AsyncExec Activity
Activity
Workflow
GetServerContext
Controller methods
800
600
400
200
0
R34
R35
R36
R37
R38
R39
R39D
R40
R40C
R41
R41D
R42
R43
R44
R45
R46
7/1/2023
1/1/2024
7/1/2024
1/1/2025
7/1/2025
1/1/2026
7/1/2026

Join the OSS Adopter Program!

Provision **Operate**

with Maintainers

You get a great experience

Everybody's experience improves!

Learn More / Join the Community

KubeCon CloudNativeCon North America 2026

- CNCF Slack (main communication channel)
  - **#cadence-users**
  - **#cadence-contributors**

- **cadenceworkflow.io**
  (main website: blogs, wikis, get-started, …)

- /cadenceworkflow on LinkedIn
  (primary channel for updates and announcements)

- @cadence-workflow on YouTube
- @cadenceworkflow on X
- r/cadenceworkflow on Reddit



KubeCon CloudNativeCon North America 2026

Thank You!



Ender Demirkaya
Uber, Cadence
https://www.linkedin.com/in/enderdemirkaya



Soma Koltai
Cloudera
https://hu.linkedin.com/in/koltaisoma